Load Balancing and Efficient Memory Usage for Homogeneous Distributed Real-Time Embedded Systems

Omar Kermia, Yves Sorel

Load Balancing and Efficient Memory Usage for Homogeneous Distributed Real-Time Embedded Systems

Omar Kermia, Yves Sorel

INRIA Rocquencourt,
BP 105 - 78153 Le Chesnay Cedex, France
Phone: +33 1 39 63 52 60 - Fax: +33 1 39 63 51 93
omar.kermia@inria.fr, yves.sorel@inria.fr

Abstract

This paper deals with load balancing and efficient memory usage for homogeneous distributed real-time embedded applications with dependence and strict periodicity constraints. Most of load balancing heuristics tend to minimize the total execution time of distributed applications by equalizing the workloads of processors. In addition, our heuristic satisfies dependence and strict periodicity constraints which are of great importance in embedded systems. However, since resources are limited some tasks distributed onto a processor may require more data memory than available. Thus, we propose a fast heuristic achieving both load balancing and efficient memory usage under dependence and strict periodicity constraints. Complexity and theoretical performance studies have showed that the proposed heuristic is respectively efficient and fast.
Thus, an efficient memory usage is also necessary, especially in embedded systems where memory is limited.

Although the total execution time of tasks is minimized

some tasks could not be executed because the processors where they were distributed do not own enough memory to store the data used by these tasks.

However, memory usage plays a significant role in determining the applications performances.

Keywords
Load Balancing; Multiprocessor Real-time scheduling; Memory Optimization

1 Introduction

Distributed real-time embedded applications found in domains such as avionics, automobiles, autonomous robotics, telecommunications lead to non-preemptive distributed scheduling problems with dependence and strict periodicity constraints. These complex applications must meet their real-time constraints, e.g. deadlines equal to the periods of the tasks, otherwise dramatic consequences may occur. Then, their total execution time or the completion time of the last task must be minimized in order to decrease the delay in the feedback control occurring in these applications, and in addition the memory resources must be efficiently used since they are limited due to the embedded feature. Here, because we deal with signal processing and automatic control applications the tasks have a strict periodicity. A strict period means that if the periodic task a has period T_a then $\forall i \in N, (s_{a_{i+1}} - s_{a_i}) = T_a$, where a_i and a_{i+1} are the i^{th} and the $(i+1)^{th}$ instances of the task a, and s_{a_i} and $s_{a_{i+1}}$ are their start times [1].

The problem of load balancing started to emerge when distributed memory processing was gaining popularity. Load balancing aims at decreasing the total execution time of distributed (parallel) computation by equalizing the workloads of processors during or after the distribution and the scheduling of the application. For an overview on the general load balancing problem see [2]. Since memory must be carefully managed in embedded systems we propose a new load balancing technique that provides efficient memory usage.

Studies on general purpose distributed applications showed that over 65% of processors are idle at any given time [3]. It means that some processors are underloaded when others are overloaded. This is the main reason to achieve load balancing which in addition induces a smaller total execution time than if the load is unbalanced. In our case since we aim at real-time applications, this percentage can be more important due to periodicity constraints. On the other hand since we address embedded systems we must take into account the limited memory of every processor. Thus, it is important to efficiently use the memory. This issue is much more important when, as it is the case in this paper, tasks with different periods communicate. This

is illustrated in a simple example with two tasks a and b which communicate while their periods are not the same (e.g. period of b be equal to n times the period of a) such that a distributed onto processor $P1$ produces data for b distributed onto $P2$ assuming that b depends of a. Dependent tasks scheduled onto different processors lead to inter-processor communications between these processors. We consider that before executing b the processor $P2$ must receive the n data produced by the n instances of a [4] executed by $P1$. The scheduling of these tasks is depicted in figure 1 for $n = 4$ where four data produced by the four executions of task a on $P1$ must be stored on $P2$ until b is able to use the four data, i.e. on $P2$ the memory used to store the data produced by the first instance of a cannot be reused by the data produced by the second, the third and the fourth instances of a. That means memory reuse [5] is not always possible in this case where as much data as instances are used.

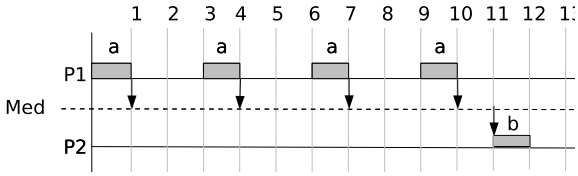

Figure 1. Data Transfer

We assume that the distributed architecture is homogeneous, i.e. the processors and the communication media are identical, and moreover they have the same memory capacity. Also, we assume that load balancing is achieved off-line in order to minimize its impact at run time. We assume inter-processor communications take time, and thus we shall take them into account in the distributed scheduling as well as in the load balancing of tasks. Since we have dependence and strict periodicity constraints, when load balancing is intended, a task is moved only if the constraints still remain satisfied. The distributed scheduling problem in such conditions is very difficult to solve. Therefore, we first perform a separate distributed scheduling heuristics [4, 6] which seeks only to satisfy the dependence and strict periodicity constraints. From this result we perform another heuristic for load balancing and efficient memory usage. We assume that each task has a known execution time (Worst Case Execution Time WCET) and a known required memory amount which represents its need in terms of data storage. Using these two information, the proposed heuristic computes a cost function to determine at any time to which processor the tasks have to be moved. It begins by grouping tasks scheduled onto the same processor into blocks according to their dependences. Finally, it distributes each resulting block onto the processor according to the value of the cost function.

We are concerned by fast heuristics because realistic industrial applications we deal with, are very complex, i.e. several thousands of tasks and tens of processors, preventing the utilization of slow heuristics. That leads us to develop a fast sub-optimal heuristic which performs load balancing and efficient memory usage for such applications.

The rest of the paper is organized as follows: the next section is devoted to the related work. Section 3 introduces some notations, gives the principles of the proposed heuristic and its pseudo code, then the proposed heuristic is illustrated by an example. In section 4, the complexity of our heuristic is studied. A theoretical performance study is proposed in section 5. Finally, Section 6 presents a conclusion.

2 Related Work

Optimal load balancing which consists in finding the smallest total execution time is an NP-hard problem [7]. Optimal algorithms for this problem are usually based on the Branch and Bound principle. Korf in [8] gives an optimal Branch and Bound algorithm for the Bin Packing problem which is similar to the load balancing problem [7]. In addition, heuristics which produce sub-optimal solutions have been developed. For example "Genetic Algorithms" have gained immense popularity over the last few years as a robust and easily adaptable search technique. For example Greene proposed in [9] a Genetic Algorithm for load balancing of general purpose distributed applications. In these works, the memory usage was not taken into account. There exist only few load balancing algorithms which consider memory usage [10, 11]. On the other hand, the notion of "Memory Balancing" is used as in [12] which considers only memory balancing and no load balancing.

3 Load Balancing Heuristic with efficient usage of memory

3.1 Definitions

The proposed heuristic deals with applications involving N tasks and M processors. Each task a has an execution time E_a, a start time S_a computed by the distributed scheduling heuristic, and a required memory amount m_a. The required memory amount may be different for every task. It represents the memory space necessary to store the data managed by the task, i.e. all the variables necessary for the task according to their types.

As shown in [13] to analyze an application composed of periodic tasks it is enough to study its behavior for a time interval equal to the least common multiple (LCM)

of all the task periods, called the hyper-period. Because of the strict periodicity constraints, each task is repeated according to the ratio of the hyper-period and its period. This ratio corresponds to the number of instances of the task on the hyper-period. The time elapsed between the start times of two successive instances is always equal to its period [4]. For each processor, the proposed heuristic considers all the tasks scheduled in a time interval equal to $[S_0^P, S_0^P + LCM]$, where S_0^P is the start time of the first task scheduled onto P.

The inter-processor communication times are taken into account by the proposed heuristic according to the following principle. When a task is scheduled onto a processor P, if there is a dependence between this task and n ($n \in N$, $n \geq 1$) other tasks already scheduled onto other processors, n new receive tasks must be created and scheduled before this task in order to receive the data on that processor P. On the other hand a send task must be created and scheduled onto the processor where the producer task is scheduled. The data transfer associated to a dependence is carried out by sending and receiving messages through the communication medium which connects both processors where corresponding tasks are executed. The communication time specifies the time elapsed between the start time of the sending task and the completion time of the receiving task. Since the communication time depends on the size of data to be transferred (the larger the task, the longer the transfer time), the memory usage affects communication times [14].

In multi-periodic applications, the data transfer between the tasks is not achieved by the same way as in the non-periodic applications. When there is a dependence between task a and b and the period of b is twice the period of a, task a is repeated twice as fast as task b. It means that task b needs two data produced by task a to be executed (the data produced by each execution of a are generally different). This principle explains the possible dependence between tasks at different periods knowing that the problem does not exist when they are at the same period. Here is a simple example of that. Let a be a sensor which measures the temperature of an engine, and let b be the task which computes the average temperature of the same engine (period of b is equal to n times the period of a). Therefore a is repeated n times before executing b which has to receive n data from a to compute the average temperature.

Tasks are grouped into blocks according to the following principle. A block is built of one task or several dependent tasks scheduled onto the same processor such as the move of one of these tasks produces an inter-processor communication. A block moves from its initial processor to another processor, likewise, and in order to keep the same designation a block can also move from its initial processor to this same processor.

When a block moves, either it keeps the same start time, or its start time decreases leading to decrease the total execution time. The execution time (resp. the required memory amount) of a block is the sum of the execution times (resp. the required memory amounts) of the tasks it contains, and its start time is the start time of the first task.

Let B be the block containing the tasks $\{b_0, .., b_i, .., b_n\}$ scheduled onto the same processor. Let a and c be two tasks scheduled onto the same processor where B is scheduled, such as $a \prec b_i$ and $b_i \prec c$ ($a \prec b_i$ means that there are dependences between a and b_i, i.e. b_i cannot start until a is completed). Let E_a be the execution time of a, and C the communication time. Then, B is a block if

$$\forall i \in N, 0 \leq i \leq n, \quad S_{b_i} \geq (S_a + E_a + C) \tag{1}$$

and

$$(S_{b_i} + E_{b_i} + C) \leq S_c \tag{2}$$

Equation (1) deals with the dependence $a \prec b_i$ and equation (2) deals with the dependence $b_i \prec c$.

We distinguish two categories of blocks:

1. *a block whose tasks are only the first instances of each of these tasks. These blocks contain only first instances of tasks. This category represents blocks which can improve the total execution time because their start times can decrease when they are moved from a processor to another one. The other instances of these tasks belong to blocks of the second category.*

2. *a block whose the first task is another instance than the first instance of this task. The other tasks of this category of blocks are either the first or other instances of tasks. The start time of this category of blocks decreases only if the start time of the first instance of its first task which belongs to a block of the first category decreases.*

We denote by $G_{P_i \to P_j}(A)$ the gain in terms of time due to the move of the block A of the first category from the processor P_i to a processor P_j. P_j may be the same or different than P_i. $S_A^{P_i}$ is its initial start time and $S_A^{P_j}$ is its new start time on P_j. $S_A^{P_j}$ is less or equal to $S_A^{P_i}$.

$$G_{P_i \to P_j}(A) = S_A^{P_i} - S_A^{P_j} \tag{3}$$

When $S_A^{P_j} = S_A^{P_i}$, $G_{P_i \to P_j}(A) = 0$

As explained before, on each processor, tasks are repeated according to their periods inside a time interval equal to the hyper-period and they are scheduled upon that hyper-period which is repeated infinitely. In order to guarantee that repetition we introduce a condition, called Block

Condition, *inside the load balancing heuristic. It consists in checking that before moving a block to a processor it does not prevent the execution of the next instance of the first block moved to this processor.*

Let assume A is the first block which has been moved to processor P. A block B satisfies the Block Condition *if*

$$S_B^P + E_B \leq S_A^P + LCM \quad (4)$$

LCM is the least common multiple of all periods of tasks.

The heuristic is based on a Cost Function $\lambda_{P_i \to P_j}(A)$ which is computed for a block A initially scheduled on P_i and a processor P_j. It combines $G_{P_i \to P_j}(A)$ and the sum of required memory amounts by the k blocks $B_1, ..., B_k$ already moved to this processor P_j.

$$\lambda_{P_i \to P_j}(A) = \begin{cases} G_{P_i \to P_j}(A) & \text{if no block has been moved to } P_j \\ \frac{G_{P_i \to P_j}(A)+1}{\sum_{i=1}^{i=k} m_{B_i}} & \text{otherwise} \end{cases} \quad (5)$$

1 is added to $G_{P_i \to P_j}(A)$ since $G_{P_i \to P_j}(A)$ may take the value 0.

When the gain $G_{P_i \to P_j}(A)$ is maximized and $\frac{1}{\sum_{i=1}^{i=q} m_{B_i}}$ is maximized, i.e. the required memory amount is minimized, thus $\lambda_{P_i \to P_j}(A)$ is maximized.

3.2 The Proposed Heuristic

For each processor the proposed heuristic starts by building blocks from tasks distributed and scheduled onto this processor. Then, each block A is processed according to the increasing order of their start times. This process consists in computing the cost function λ for the processors whose end time of the last block scheduled on these processors are less or equal than the start time of the block A, and in seeking the processor which maximizes λ. Moreover, a block is moved to that processor if the LCM condition *is verified, otherwise that processor is no longer considered, and the heuristic seeks again another processor which maximizes λ. If the moved block belongs to the first category and $\lambda > 0$, then this block will decrease its start time. In order to keep its strict periodicity constraint satisfied, the heuristic looks through the remaining blocks and updates the start times of the blocks containing tasks whose instances are in the moved block (see step 3 in the example of section 3.3). Algorithm 3.2 details the different steps of the proposed heuristic.*

[ht!] Load Balancing heuristic [1] Each processor P_i Build the blocks by grouping tasks Sort the blocks by their start times in an increasing order Each block A_i initially scheduled on P_i Each processor P_j Compute the cost function $\lambda_{P_i \to P_j}(A_i)$ for each processor P_j whose end time of its last block is less or equal to S_{A_i} Seek the processor P which maximizes $\lambda_{P_i \to P}(A_i)$ and verify that the start time of A_i satisfies the LCM condition*, otherwise seek another processor $G_{P_i \to P}(A_i) > 0$ on P Update the start times of the blocks containing tasks whose instances are in A_i Move A_i to P*

3.3 Basic Example

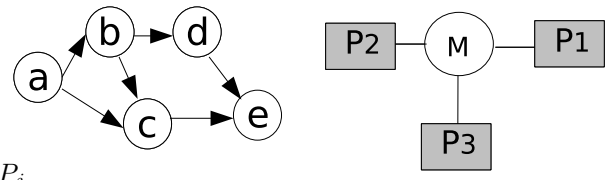

Figure 2. Task graph and architecture

In order to illustrate how the proposed heuristic progresses we first applied the distributed scheduling heuristic given in [4] to the system of figure 2. The periods of the tasks are: T_a=3 units, T_b=6 units, T_c=6 units and T_d=T_e=12 units. The architecture is composed of three identical processors $P1, P2, P3$ connected by a medium Med. The execution times of all the tasks are: E_a=E_b=E_c=E_d=E_e=1 unit. The communications times are: C=1 unit. The required memory amount of the tasks are: $m_a = 4$ units, $m_b = m_c = 1$ unit, $m_d = m_e = 2$ units. The result is depicted on figure 3. Figure 3 shows that the total execution time is 15 units. The sum of required memory amount of tasks scheduled onto P_1 is 16 units, this sum in P_2 is 4 and 4 in P_3. Each task a_i constitutes a block, tasks b_j, c_j form the blocks [$b_1 - c_1$], [$b_2 - c_2$] and tasks d, e form the block [$d - e$]. Then, following the increasing order of the block start times, they are moved to the processors as follows (the first three steps are fully detailed):

1. *block $[a_1]$ is selected, $G_{P_1 \to P_1}([a_1]) = G_{P_1 \to P_2}([a_1]) = G_{P_1 \to P_3}([a_1]) = 0$ and $\sum_{i \in p_2} m_i = \sum_{i \in p_3} m_i = \sum_{i \in p_1} m_i = 0$ so $\lambda_{P_1 \to P_1, P_2, P_3}([a_1]) = 0$. We choose to keep $[a_1]$ scheduled onto P_1,*
2. *block $[a_2]$ is selected, $G_{P_1 \to P_1}([a_2]) = G_{P_1 \to P_2}([a_2]) = G_{P_1 \to P_3}([a_2]) = 0$ and $\sum_{i \in p_2} m_i = \sum_{i \in p_3} m_i = 0$, $\sum_{i \in p_1} m_i = 4$ so $\lambda_{P_1 \to P_1}([a_2]) = 1/4$, $\lambda_{P_1 \to P_2, P_3}([a_2]) = 1$. We choose to move $[a_2]$ to P_2 because $\lambda_{P_1 \to P_2, P_3}([a_2]) > \lambda_{P_1 \to P_1}([a_2])$ (P_3 could be chosen also);*

3. *block* $[b_1 - c_1]$ *is selected,* $G_{P_2 \to P_1}([b_1 - c_1]) = G_{P_2 \to P_3}([b_1 - c_1]) = 0$, $G_{P_2 \to P_2}([b_1 - c_1]) = 1$ *and* $\sum_{i \in p_2} m_i = \sum_{i \in p_1} m_i = 4$, $\sum_{i \in p_3} m_i = 0$ *so* $\lambda_{P_2 \to P_3}([b_1 - c_1]) = 0$, $\lambda_{P_2 \to P_1}([b_1 - c_1]) = 1/4$, $\lambda_{P_2 \to P_2}([b_1 - c_1]) = 1/2$, *$[b_1 - c_1]$ is moved to P_2. As $[b_1 - c_1]$ is a block of the first category and $\lambda > 0$ then the block $[b_2 - c_2]$ which is a block of the second category decreases its start time from 11 to 10 units;*

4. *block* $[a_3]$ *is selected,* $\lambda_{P_1 \to P_1}([a_3]) = 1/4$, $\lambda_{P_1 \to P_2}([a_3]) = 1/6$ *and* $\lambda_{P_1 \to P_3}([a_3]) = 1$, *$[a_3]$ is moved to P_3;*

5. *block* $[a_4]$ *is selected,* $\lambda_{P_1 \to P_1}([a_4]) = 1/4$, $\lambda_{P_1 \to P_2}([a_4]) = 1/6$ *and* $\lambda_{P_1 \to P_3}([a_4]) = 1/4$, *$[a_4]$ is moved to P_1;*

6. *block* $[b_2 - c_2]$ *is selected,* $\lambda_{P_2 \to P_1}([b_2 - c_2]) = 1/8$, $\lambda_{P_2 \to P_2}([b_2 - c_2]) = 0/6 = 0$, $\lambda_{P_2 \to P_3}([b_2 - c_2]) = 0/4 = 0$, *$[b_2 - c_2]$ is moved to P_1;*

7. *block* $[d - e]$ *is selected,* $\lambda_{P_3 \to P_1}([d - e]) = 1/10$ *on* P_1 *but it does not satisfy the LCM condition,* $\lambda_{P_3 \to P_2}([d - e]) = 1/6$, $\lambda_{P_3 \to P_3}([d - e]) = 1/4$, *$[d - e]$ is moved to P_3.*

Finally, all the blocks are moved and we obtain a new distribution and scheduling depicted on figure 4. The first observation is that the total execution time is now 14 *units instead of* 15. *Secondly, the required memory amount was* $[P_1 : 16, P_2 : 4, P_3 : 4]$, *the memory amount the heuristic provides is:* $[P_1 : 10, P_2 : 6, P_3 : 8]$ *that corresponds to a better memory usage.*

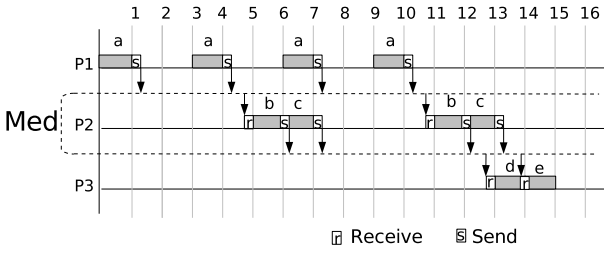

Figure 3. Scheduling before load balancing

4 Complexity study

Let N_{blocks} *be the number of blocks built from* N *tasks. The complexity of our heuristic is* $O(MN_{blocks})$, *where* $N_{blocks} \leq N$.

Since usually the number of sensors which impose their periods to the tasks is small [15], the number of different periods is small. Moreover, as the dependent tasks which are at the same or multiple periods are scheduled onto the same processor [4], the number of blocks is small, even though each block may include a large number of tasks.

The polynomial complexity of the heuristic as well as the small number of blocks ensure a fast execution time.

The value of N_{blocks} *is can be very small relatively to* N *because the number of different periods among the multi-periodic applications tasks due to the relatively small number of sensors and actuators which impose their periods to the tasks [15]. As the dependent tasks are at the same or multiple periods they are scheduled onto the same processor [4] leading to a relatively small number of blocks but which includes a large number of tasks. Therefore, the proposed heuristic has a fast execution time.*

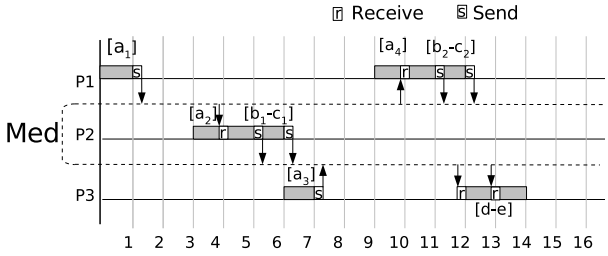

Figure 4. Scheduling after load balancing

5 Theoretical performance study

In order to assess the theoretical performance of our heuristic we calculate upper and lower bounds for the total execution time, and calculate an α*-approximation for the memory usage.*

5.1 Total execution time bounds

If L_{former} *is the total execution time before applying the load balancing heuristic and* L_{new} *the total execution time after applying the heuristic then:*

$$G_{total} = L_{former} - L_{new}$$

Let assume that in the architecture each two processors are connected by a communication medium (one medium can connect several processors pairs such as in the architecture depicted in the example figure 2). The following theorem introduces the upper and the lower bounds which restrict G_{total}. *These bounds allow us, on one hand, to be sure that* L_{new} *is always less or equal to* L_{former} *and, on the other hand, to know how much* G_{total} *may at most be improved.*

Theorem 1

The value of G_{total} is bounded by:

$$0 \leq G_{total} \leq \gamma[(M-1)!] \quad (6)$$

γ is a communication time. M is the number of processors in the architecture.

Proof

First, let us begin by prove that $G_{total} \leq \gamma(M-1)!$. When a block B (one or several tasks) is moved from a processor P_i to another processor P_j, the communication which connects B in P_i to a block A in P_j is suppressed. Let assume that γ is the longest communication among the whole communications that we can delete by moving blocks.

It is interesting to note that even though there exist other moves from P_i to P_j or from P_j to P_i performed before or after moving B, they do not take effect on G_{total}, i.e. the total execution time is not decreased of the sum of all these communications times, because the communication between A and B includes all the other communications which have a time smaller than its communication time. This is due to the fact that the blocks are scheduled sequentially on a processor, and if the start time of a block decreases all the start times of the blocks scheduled after it, decrease also. Consequently, the total execution time is decreased at most of: $\gamma \times$ [number of processors pairs]. The number of distinct processors pairs is equal to $(M-1)!$.

Second, let us prove that $0 \leq G_{total}$. The proposed heuristic tends to move every block to a processor such that its start time decreases, or at worst the block keeps its initial start time. Likewise, blocks which communicate with a moved block will either decrease their start times or keep their initial start times. It implies that G_{total} is at least equal to 0.

Hence, (6) is proven

The previous theorem shows on the one hand that, in some cases all the communications can be suppressed and the total gain is equal to $\gamma(M-1)!$ but this is quite rare, and on the other hand that our heuristic never increases the total execution time.

5.2 α-approximation for the memory usage

Here we only consider memory, notice that the total execution time is not taken into consideration. The cost function in this case is equal to $\lambda = \frac{C_{st}}{\sum_{i=1}^{k} m_{B_i}}$ where C_{st} is a constant number (we assume that the gain G is a constant). Thus, for each block the heuristic chooses the processor which maximizes λ, i.e. the processor which minimizes $\sum_{i=1}^{k} m_{B_i}$ (k is the number of blocks B_i already moved, see the last part of section 3.1).

5.2.1 α-approximation definition

Let us consider an arbitrary optimization problem. Let OPT(X) denotes the value of the optimal solution for a given input X, and let A(X) denotes the value of the solution computed by algorithm A using the same input X. We say that A is an α-approximation algorithm, of minimization, type if $\frac{A(X)}{OPT(X)} \leq \alpha$ for all inputs X. A 1-approximation algorithm always returns the exact optimal solution. The approximation factor α may be either a constant or a function of the inputs. A more detailed definition of approximation methods and presentations of the problem that have been approximatively resolved using these algorithms can be found in [16].

5.2.2 Heuristic α-approximation

In this section we introduce a theorem which proves that the proposed heuristic is $(2 - \frac{1}{M})$-approximated. This result gives an idea of the proposed heuristic performance when it deals with memory usage only.

Theorem 2

Our heuristic is a $(2 - \frac{1}{M})$-approximation algorithm (we remind that M is the number of processors). More precisely, if ω_{opt} is the optimal solution and ω is the solution obtained by the proposed heuristic then:

$$\frac{\omega}{\omega_{opt}} \leq 2 - \frac{1}{M} \quad (7)$$

ω_{opt} and ω are the maximal memory amount used in one processor among all memory amounts used in all architecture processors.

Proof

Let us consider that a block A is moved to processor P_i with a required memory amount m_A. We denote by V the required memory amount for blocks moved to the processor P_i before moving the block A, this memory amount is the smallest among all the processors. This is why P_i is chosen by the proposed heuristic in order to move the block A to this processor. If ω_{opt} is the optimal solution then m_A which is only a part of the solution is such that

$$m_A \leq \omega_{opt} \quad (8)$$

If N_{blocks} is the number of blocks formed by applying the heuristic then,

$$\frac{\sum_{i=1}^{N_{blocks}} m_{blocki}}{N_{blocks}} \leq \omega_{opt} \quad (9)$$

From the definition of V we have

$$V + \frac{m_A}{M} \leq \frac{\sum_{i=1}^{N_{blocks}} m_{block_i}}{N_{blocks}} \quad (10)$$

From (9) and (10) we have

$$V + \frac{m_A}{M} \leq \omega_{opt} \quad (11)$$

Thus, the required memory amount on P_i after moving A to P_i is

$$\omega = V + m_A \tag{12}$$

By rewriting ω we have

$$\omega = V + \frac{m_A}{M} + m_A\frac{M-1}{M} \tag{13}$$

From (8) and (11) we have

$$V + \frac{m_A}{M} + m_A\frac{M-1}{M} \leq (1 + \frac{M-1}{M})\omega_{opt}$$

From (13) we have

$$\omega \leq (1 + \frac{M-1}{M})\omega_{opt}$$

Thus

$$\frac{\omega}{\omega_{opt}} \leq 2 - \frac{1}{M}$$

Hence (7) is proven

6 Conclusion

In this paper we presented a heuristic for load Balancing and efficient memory usage of homogeneous distributed real-time embedded systems. Such systems must satisfy dependence and strict periodicity constraints, and their total execution time must be minimized since they include feedback control loops. In addition since memory is limited in embedded systems it must be efficiently used. This is achieved by grouping the tasks into blocks, and moving them to the processor such that the block start time decreases, and this processor has enough memory capacity to execute the tasks of the block. We shown that the proposed heuristic has a polynomial complexity which ensures a fast execution time. We performed also a theoretical performance study which bounds the total execution time decreasing, and shows that our heuristic is a $(2 - \frac{1}{M})$-approximation algorithm for the memory usage, with M the number of processors.

Presently, the proposed heuristic was not yet applied on a realistic application, but the results obtained from the theoretical analysis demonstrated the effectiveness of the proposed algorithm compared to an optimal algorithm. This α-approximation analysis is widely used and has the advantage to avoid the waste of time in implementing and testing a heuristic if the analysis demonstrates that it is not effective.

We plan to implement our proposed heuristic in a CAD software for real-time systems and experiment it.

References

[1] *L. Cucu and Y. Sorel. Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints. In* Proceedings of the 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group, PLANSIG'04*, Cork, Ireland, December 2004.*

[2] *B. A. Shirazi, K. M. Kavi, and Ali R. Hurson, editors.* Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.*

[3] *M.W. Mutka. Estimating capacity for sharing in a privately owned workstation environment.* IEEE Transactions on Software Engineering*, 18(4):319–328, 1992.*

[4] *O. Kermia and Y. Sorel. A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor. In* Proceedings of ISCA 20th international conference on Parallel and Distributed Computing Systems, PDCS'07*, Las Vegas, Nevada, USA, September 2007.*

[5] *S. Biswas, T. Carley, M. Simpson, B. Middha, and R. Barua. Memory overflow protection for embedded systems using run-time checks, reuse, and compression.* Trans. on Embedded Computing Sys.*, 5(4):719–752, 2006.*

[6] *C. Hou and K. Shin. Allocation of periodic tasks modules with precedence and deadline constraint in distributed real-time systems. volume 46, December 1997.*

[7] *C. Ekelin and J. Jonsson. A lower-bound algorithm for load balancing in real-time systems. In* Proceedings of the International Conference on Principles of Distributed Systems (OPODIS'03)*, La Martinique, France, December 2003.*

[8] *R. E. Korf. A new algorithm for optimal bin packing. In* Eighteenth national conference on Artificial intelligence*, pages 731–736, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.*

[9] *W. A. Greene. Dynamic load-balancing via a genetic algorithm. In* ICTAI '01: Proceedings of the 13th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'01)*, page 121, Washington, DC, USA, 2001. IEEE Computer Society.*

[10] *Y. Liu, T. Liang, C. Huang, and C. Shieh. Memory resource considerations in the load balancing of software dsm systems. In* ICPP Workshops*, pages 71–78, 2003.*

[11] *T. Kimbrel, M. Steinder, M. Sviridenko, and A. N. Tantawi. Dynamic application placement under service and memory constraints. In* WEA*, pages 391–402, 2005.*

[12] *K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors.* ACM Transactions on Computer Systems*, 18(3):229–262, 2000.*

[13] *K. Danne and M. Platzner. A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware. In* Proceedings of the International Conference on Field Programmable Logic and Applications*, pages 568–573, August 2005.*

[14] *Krzysztof Kuchcinski. Embedded system synthesis by timing constraints solving. In* ISSS '97: Proceedings of the 10th international symposium on System synthesis*, pages 50–57, Washington, DC, USA, 1997.*

[15] *M. Alfano, A. Di-Stefano, L. Lo-Bello, O. Mirabella, and J.H. Stewman. An expert system for planning real-time distributed task allocation. In* Proceedings of the Florida AI Research Symposium*, Key West, FL, USA, May 1996.*

[16] *C. A. S. Oliveira. Approximation algorithms for combinatorial optimization. In* C.A. Floudas and P.M. Pardalos, editors, Encyclopedia of Optimization. Kluwer Academic Publishers*, 2005.*